オープン・ソースのCPUコアの実力を試す

32ビットRISCプロセッサ「LatticeMico32」レビュー

山際伸一

ここでは，無償で利用できるソフト・マクロのCPU「LatticeMico32」を取り上げる．HDL(Hardware Description Language)ソース・コードの形で提供され，FPGA(Field Programmable Gate Array)に限らず，ASIC(Application Specific Integrated Circuit)にも実装できる．オンチップ・バスとしてWISHBONEバス・インターフェースに対応している．ソフトウェア開発環境はGNUベースのツール群を利用する．（編集部）

米国Lattice Semiconductor社は，オープン・ソースのCPUコア「LatticeMico32」注1を提供しています(p.26のコラム「LatticeMico32の入手と開発ツールのセットアップ」を参照)．LatticeMico32は，32ビット命令長/データ長のRISC(Reduced Instruction Set Computer)アーキテクチャのCPUコアです．32本の汎用レジスタを搭載し，外部割り込み機能やキャッシュ・メモリを用いることができます．さらに，ユーザが命令を定義できます．

本稿では，LatticeMico32のアーキテクチャについて解説した後，ハードウェアとソフトウェアの開発方法を説明します．例題は，UARTインターフェースとLEDを使って割り込みを確認する簡単なシステムです．

1．LatticeMico32のアーキテクチャ

LatticeMico32は，32ビット命令/データ長のRISCプロセッサ・コアです．アドレッシングには，ビッグ・エンディアンを採用しています．**図1**に機能ブロック図を示します注2．

本稿では，LatticeMico32のCPUコアとしての基本的な機能に主眼を置いて説明していきます．

● ハーバード・アーキテクチャとWISHBONEバス

LatticeMico32は，命令とデータのパスがそれぞれ独立しているハーバード・アーキテクチャを採ります．それぞれのバスも独立しており，WISHBONEインターフェース(p.27のコラム「WISHBONEインターフェース」を参照)で外部に接続されています．それぞれのバスに独立したメモリを接続することにより，命令フェッチとデータ・アクセスの衝突を回避できます．

命令バスから読まれた命令コードは，命令レジスタにラッチされ，デコードされます．デコード時に，その命令が用いる読み出し元データ・パスや，書き込み先データ・パス，演算器を選択し，命令を実行します．

演算器としては，加算器(減算もここで行われる)や論理演算器，シフタ，乗算・除算器が用意されています．シフタと乗算・除算器に関しては，複数サイクルで実行します．これらの演算器は，プロセッサの設定時に性能を選択できます．しかし，高速な演算器は多くのハードウェアを使う

注1：LatticeMico32のWebサイトは，http://jp.lscc.com/products/intellectualproperty/ipcores/mico32/index.html

注2：LatticeMico32開発システムのインストール先のフォルダにLatticeMico32プロセッサ・アーキテクチャに関するPDFファイルが保存されている．デフォルトのパスにインストールした場合は，C:\LatticeMico32\micosystem\components\lm32_top\document\lm32_archman.pdfがそのリファレンス・マニュアル．このドキュメントがLatticeMico32プロセッサを用いたシステム開発に必要な情報で最も詳しい資料である．

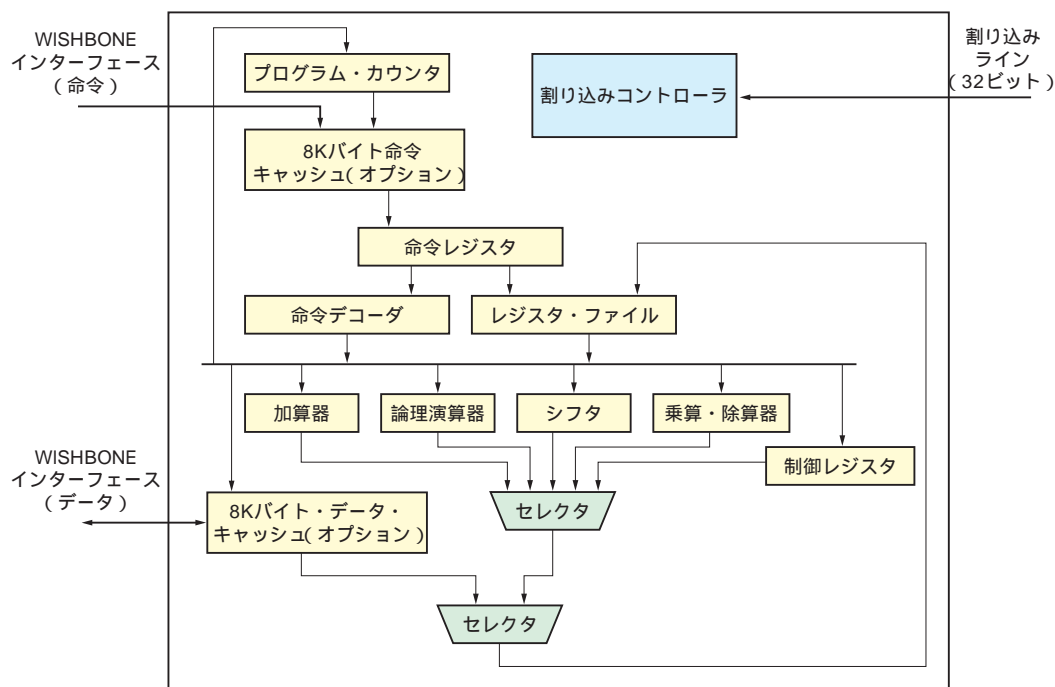
KeyWord　ソフト・マクロ，LatticeMico32，FPGA，ASIC，CPUコア，RISC，WISHBONE，ハーバード・アーキテクチャ，分岐予測，例外ハンドラ，Spartan-3E，Cyclone II

ので，性能と規模のトレードオフになります．演算結果は，それぞれの演算器の出力から一つ選択され，書き込み先に書き込まれていきます．

● キャッシュ・メモリを設定可能

LatticeMico32には，命令とデータに対してキャッシュ・メモリを追加できます．多くのメモリを利用する用途では，キャッシュ・メモリが効果を発揮します．しかし，ターゲットとなるFPGAが持つメモリ容量が小さかったり，キャッシュ・サイズ前後のメモリ空間しか利用しなかったりする場合は，意味がないかもしれません．性能と必要となるハードウェア・リソースのトレードオフを十分に検討する必要があります．

キャッシュ・メモリは，ライト・スルー・キャッシュで，

図1 LatticeMico32の機能ブロック図
32ビット命令/データ長のRISCプロセッサ・コアである．命令とデータのパスがそれぞれ独立しているハーバード・アーキテクチャを採る．オンチップ・バスはWISHBONE．

コラム LatticeMico32の入手と開発ツールのセットアップ

Lattice Semiconductor社のWebサイト(`http://www.latticesemi.co.jp/`)の「アカウント・インフォ」をクリックして開くページで，アカウントを作成しておきます．そして，「ラティスMico32」のページから「ラティスMico32システムをダウンロード」のリンクをクリックします．ライセンス条項に同意すると開発システムのインストーラをダウンロードできます．

インストールの際，GNU-based Compiler Toolsのチェックを外しておいてください．今回はCygwinで開発するのでGNUツールのインストールは必要ありません．

デフォルトのパスにLatticeMico32開発システムをインストールした場合，`C:\LatticeMico32\micosystem\components`にハードウェアのソース・コードがインストールされます．プロセッサはlm32_topフォルダにあります．関連するハードウェアはすべてVerilog HDLで記述されています．

ソフトウェア開発ツールについては，前述の「ラティスMico32」のページにある「Downloadable Software」のリンクをクリックし，表示されたツールの一覧からLatticeMico32 GNU-Tools Source Codeをダウンロードします．Mico32プロセッサのソフトウェアは，GCCで開発できます．

本稿では，バージョン6.1.1のGNUツールを用います．ソース・コードで配布されるので，Cygwin向けにコンパイル済みのものを本誌CD-ROMに掲載します．http://www.cygwin.com/を参照してCygwin環境をセットアップしておいてください．

付属CD-ROMに収録のコンパイル済みツール・チェーンは，Cygwinコンソールから以下の手順で/usr/localディレクトリに展開します．

```
$ cd /usr/local
$ tar jxvf lm32-tools.tar.bz2
```

これで，/usr/local/lm32-tools/binにgccなどのコマンドが展開されます．

コラム WISHBONEインターフェース

LatticeMico32は，バスにWISHBONEインターフェースが使われています．WISHBONEインターフェースは，フリーのIPコアを提供しているOpenCoresサイト（http://www.opencores.org/）で推奨する標準バス規格です．

WISHBONEのデータ線やアドレス線，制御線は単方向で，図B-1のように接続されます．

ADR_I/ADR_O信号は，バス・サイクルで読み書きされるメモリのアドレスです．

DAT_I/DAT_O信号は，バス・サイクルで読み書きされるデータです．

WE_I/WE_Oは書き込みイネーブル信号です．“H”で書き込みサイクル，“L”で読み出しサイクルを表現します．

SEL_I/SEL_O信号はデータ・レーンのイネーブル信号です．LatticeMico32では，4ビットが割り当てられています．つまり，バイト・イネーブルとして用いられます．

STB_I/STB_O信号はストローブ信号です．スレーブ側の入力に“H”が入力されると，そのスレーブが反応しなければなりません．WISHBONEインターフェースでは，周辺機能のメモリ・マップをこのストローブ信号を制御することで実現します．つまり，ADR_I/ADR_O信号をデコードし，STB_I/STB_O信号を作成し，スレーブ側に入力することで，アドレスごとに反応する機能を区別できるようになります．

ACK_I/ACK_O信号は，サイクルの完了を示す信号です．この信号が“H”になったサイクルがトランザクションの最後になります．読み出しサイクルでは，スレーブが読み出しデータをDAT_I/DAT_O信号に出力したサイクルで，ACK_I/ACK_O信号が“H”になります．書き込みサイクルでは，スレーブ側のメモリに書き込みが完了したサイクルで“H”になります．

CYC_I/CYC_O信号は，バス・トランザクションが行われている間“H”になります．この信号を参照することで，WISHBONEインターフェースがアクティブかどうかを知ることができます．

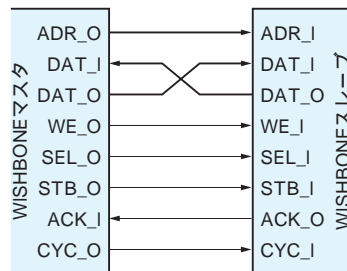
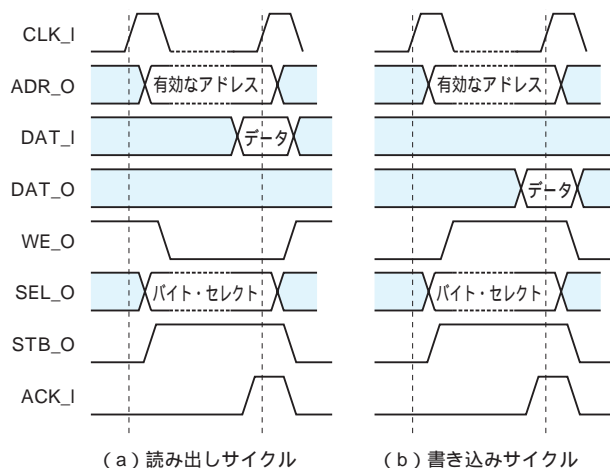
以上の信号線を元に，WISHBONEインターフェースのマスタから見た読み出し・書き込みトランザクションの例を図B-2に示します．バス・トランザクションはSTB_I/STB_O信号が“H”になることで開始しているのが分かります．その間，ADR_I/ADR_O信号にマスタは有効なアドレスを，SEL_I/SEL_O信号に有効なバイト・セレクトを出力しなければいけません．読み出しまたは書き込みが完了すると，ACK_I/ACK_O信号が“H”になりトランザクションを終了しているのが分かります．

LatticeMico32では，WISHBONEの基本的な信号線に加え，複雑なバス制御を行う信号線も用いられています．

RTY_I/RTY_O信号はスレーブがバス・サイクルに反応できないときに“H”になります．

ERR_I/ERR_O信号はスレーブがバス・サイクルを異常終了したときに“H”になります．

CTI_I/CTI_O信号はバス・サイクルのタイプを示します．

BTE_I/BTE_O信号はバースト・サイクルの際に，バーストのタイプを示します．CTI_I/CTI_O信号とBTE_I/BTE_O信号の値を表B-1に示します．

図B-1　WISHBONEの接続

図B-2　WISHBONEインターフェースの読み書きサイクル
WISHBONEは，OpenCoresの標準バスである．素直で単純なプロトコルであることが分かる．

表B-1　CTI_I/CTI_O信号とBTE_I/BTE_O信号の値

(a) CTI_IO信号の値

値	意　味
00	リニア・バースト
01	4ビート・バースト
10	8ビート・バースト
11	16ビート・バースト

(b) BTE_IO信号の値

値	意　味
000	シングル・データ・アクセス
001	アドレス固定バースト・アクセス
010	アドレス・インクリメント・バースト・アクセス
111	バースト・アクセスの終了

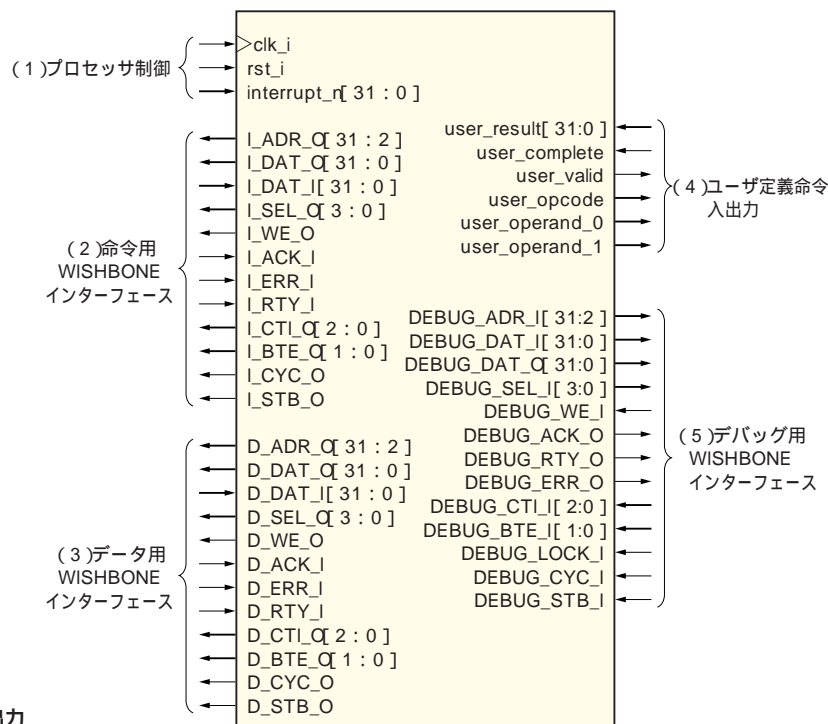



図2
LatticeMico32の入出力

32Kバイトまで2の累乗のKバイト数が設定できます．また，アソシアティブの数，セット数，ライン・サイズを設定できます．

● 32個の割り込み入力を持つ

LatticeMico32は，32ビットの割り込みラインを持ちます．つまり，32個の割り込み入力を判断できます．

● 五つの入出力インターフェース

入出力インターフェースを**図2**に示します．機能的に大きく次の五つに分かれます．

(1) プロセッサ制御

プロセッサ制御には，アクティブ"H"のクロック入力とリセット入力，アクティブ"L"の割り込み入力などがあります．割り込み入力は32ビット・バスです．

(2) 命令用WISHBONEインターフェース

命令用WISHBONEインターフェースは，外部メモリから命令をフェッチするためのバスです．マスタとして動作します．命令用のメモリ・インターフェースに接続されます．

(3) データ用WISHBONEインターフェース

データ用WISHBONEインターフェースは，外部メモリに対してデータを読み書きするためのバスです．マスタとして動作します．データ用のメモリ・インターフェースに接続されます．

(4) ユーザ定義命令入出力

LatticeMico32では，ユーザが独自の命令を定義できます．ユーザ定義の命令が実行されると，OPコードと二つのオペランドを出力します．それらの情報を使ってuser_result入力に計算結果を返し，user_completeをアサートする回路をここに接続します．

(5) デバッグ用WISHBONEインターフェース

デバッグ用のバスを用いるように設定した場合は，コアの内部にメモリを配置します．このメモリにプロセッサの状態などを記録しておいて，デバッグ用WISHBONEインターフェースを使って，そのメモリにアクセスします．すなわち，このバス(WISHBONEインターフェース)はプロセッサ外部からアクセスされるスレーブとして動作します．

● 選択可能なオプション機能

CPUコアの構成を決定するためのオプションを**表1**に示します．Verilog HDLのソース・コードの中にある`'define`文で宣言することでこれらのオプションを有効にします(p.30のコラム「分岐予測」を参照)．

CPUコアのソース・コードでは，system_conf.vというファイルが`include`されています．このファイルに**表1**の定義を集約して記述し，作成しておく必要があります．

表1 LatticeMico32の構成オプション

例外関連	LM32_SINGLE_STEP_ENABLED	シングル・ステップ実行例外を有効にする.
	CFG_BUS_ERRORS_ENABLED	InstructionBusError例外とDataBusError例外を有効にする.
	CFG_DEBUG_ENABLED	デバッグ・ユニットを有効にする.
	CFG_TRACE_ENABLED	トレース例外を有効にする.
制御関連	CFG_CYCLE_COUNTER_ENABLED	Cycle counter(CC)を有効にする.
	LM32_EBR_REGISTER_FILE	レジスタ・ファイルを有効にする.
	CFG_EBR_NEGEDGE_REGISTER_FILE	レジスタ・ファイルを立ち下がりエッジで制御する.
	CFG_EBR_POSEDGE_REGISTER_FILE	レジスタ・ファイルを立ち上がりエッジで制御する.
	CFG_FAST_UNCONDITIONAL_BRANCH	分岐予測を有効にする.
	CFG_DRAM_ENABLED	データ・メモリをロード/ストアのターゲットとする.
	CFG_INTERRUPTS_ENABLED	外部割り込みを有効にする.
	CFG_IROM_ENABLED	命令メモリを有効にする.
	CFG_IWB_ENABLED	命令のWISHBONEインターフェースを有効にする.
キャッシュ・メモリ関連	LM32_CACHE_ENABLED	キャッシュ・メモリを有効にする.
	CFG_DCACHE_ENABLED	データ・キャッシュを有効にする.
	CFG_ICACHE_ENABLED	命令キャッシュを有効にする.
算術演算関連	LM32_MC_ARITHMETIC_ENABLED	算術演算命令を有効にする.
	LM32_BARREL_SHIFT_ENABLED	シフト命令を有効にする.
	CFG_MC_BARREL_SHIFT_ENABLED	LUTベースのシフト機能を有効にする.最大32サイクル.
	CFG_PL_BARREL_SHIFT_ENABLED	パイプライン化されたシフタを有効にする.3サイクルで計算可能.
	CFG_MC_DIVIDE_ENABLED	割り算命令を有効にする.
	LM32_MULTIPLY_ENABLED	掛け算を有効にする.
	CFG_MC_MULTIPLY_ENABLED	LUTを使った掛け算器を実装する.最大32サイクル必要.
	CFG_PL_MULTIPLY_ENABLED	パイプライン化された掛け算器を実装する.3サイクルで実行可能.
	CFG_ROTATE_ENABLED	ローテート命令を有効にする.
	CFG_SIGN_EXTEND_ENABLED	符号拡張命令を有効にする.
デバッグ関連	CFG_JTAG_ENABLED	JTAGインタフェースを有効にする.
	CFG_JTAG_UART_ENABLED	JTAG UARTを有効にする.
	CFG_HW_DEBUG_ENABLED	JATGデバッグポートを有効にする.
	CFG_ROM_DEBUG_ENABLED	デバッグ用のメモリを有効にする.
	CFG_SIZE_OVER_SPEED	ハードウェア・サイズを優先する.
	CFG_USER_ENABLED	ユーザ定義の命令を有効にする.
	DEBUG_ROM	デバッグ用メモリを有効にする.

表1のオプションのほかに,重要となる定義があります.リセット後の命令フェッチ・アドレスを指定する必要があります.rst_i入力が“H”→“L”と変化した際に,0x0番地から実行を開始するのであれば,

```
'define CFG_EBA_RESET 32'h00000000
```

とします.このアドレスは,命令用WISHBONEインターフェースに出力されるアドレスであることに注意してください.

2. ハードウェアの開発

ここでは,LatticeMico32を搭載したシステムを開発します.LatticeMico32開発システムには,プロセッサ・コアだけでなく,オープン・ソースの周辺機能も含まれています[注3].今回は,**図3**に示すように,プロセッサの動作に必須のメモリのほかに,LEDとUARTを使った簡単なシステムを例に説明します.

● システム設計ではメモリの配置に注意

外部からの入出力は,クロック入力(clk_i),リセット入力(rst_i_n),2ビットLED出力(led_out),UART送受信ポート(tx_out,rx_in)とします.このシステムのVerilog HDL記述を**リスト1**に示します.

命令バスとデータ・バスが独立していますが,命令が格納されているメモリのアドレスとデータのアドレスが重複しないように注意します.これは,ソフトウェア開発ツールがGCCベースであるためです.GCCのリンカは,同一

注3:デフォルトのパスにLatticeMico32開発システムがインストールされている場合には,以下にある,UARTとRAMインターフェースをここでは用いる.
C:\LatticeMico32\micosystem\components\asram_top\rtl\verilog
C:\LatticeMico32\micosystem\components\uart_core\rtl\verilog

アドレスに複数のメモリ領域を定義できません．今回は，命令メモリを0x0，データ・メモリを0x20000000，UARTを0x40000000，LEDを0xF0000000からそれぞれ配置します．

命令メモリは，**リスト1**の(1)のように，専用のSRAMインターフェースに直結します．データ・メモリとUARTは，WISHBONEインターフェースを共有します．**リスト1**の(2)のように，ストローブ信号をアドレスでデコードし，バス・サイクルに反応するデバイスを選択しています．また，**リスト1**の(3)のように，SRAMインターフェースの向こう側でデータ・メモリとLEDレジスタの書き込みイネーブルとデータのセレクトを行っています．

割り込みは，UARTからのUART_RxInt信号をCPUコアのinterrupt_n[0]に入力しています．この割り込み

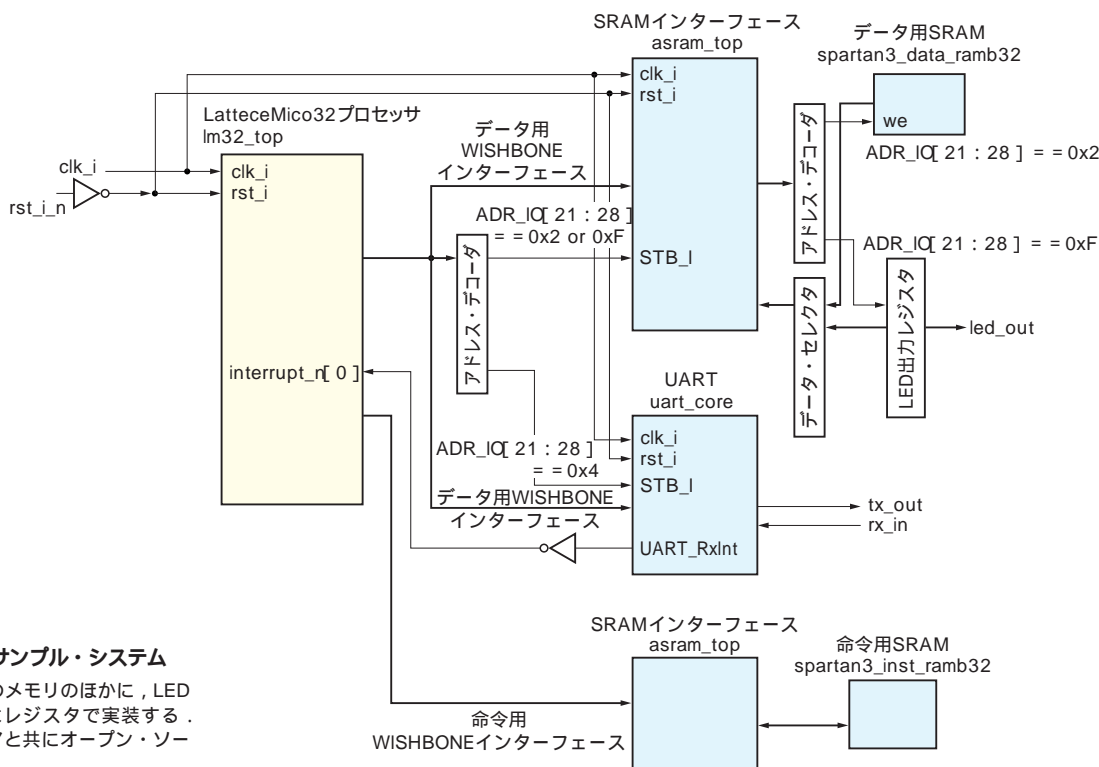



図3
LEDとUARTを使ったサンプル・システム
プロセッサの動作に必須のメモリのほかに，LEDとUARTを使う．LEDはレジスタで実装する．UARTはプロセッサ・コアと共にオープン・ソースで提供されている．

コラム　分岐予測

CPUコアの構成オプションのうち，CFG_FAST_UNCONDITIONAL_BRANCHは，分岐命令の際にその条件評価を予測することで，ループを高速化させる機能の設定です．ここでは分岐予測について説明します．

例えば，**リストC-1**のようなアセンブラで繰り返し計算する場合を考えます．

リストC-1　ループするプログラムの例

```
loop:
        ～何らかの計算～
        add       r1, r1, 1
        bg        r2, r1, loop
```

この場合，問題となるのはbg命令で，r2>r1という条件が成立するとloopラベルのアドレスに分岐します．しかし，プロセッサの動作をミクロな観点から眺めてみると，命令フェッチをするユニットは外部バス(またはキャッシュ・メモリ)からbg命令の次のアドレスの命令をロードしようとします．しかし，この計算はループですから，ほとんどの場合，条件はTRUEとなって，loopに分岐します．つまりフェッチされた命令は，プロセッサ内部で破棄され，loop番地の命令をフェッチし直すわけです．このとき，せっかく読んだ命令を破棄する操作の間，プロセッサは実行を停止させないと正しく命令実行が行われません．そのクロック・サイクルぶん，プロセッサの性能を落とすことになります．

ここで，条件は常に成立すると予測すると，無条件にloop番地を読みに行きます．成立しなかった場合はフェッチされた命令を破棄するわけですが，それは，ループの中の1回なので，その遅延は非常に少ない，と考えられます．

入力信号はアクティブ“L”です．ほかの割り込み入力のビットはすべて“H”にして，未使用としています．

ターゲットとするのは本誌2007年7月号の付属FPGA基板です．このボードに搭載されているSpartan-3Eには24Kバイトのメモリ・ブロックを内蔵していますが，命令キャッシュとデータ・キャッシュを含めると，外部メモリの内容がすべてキャッシュされてしまう状態になります．キャッシュの効果を期待できなくなるので，キャッシュは用いません．

● 命令メモリとデータ・メモリはFPGA内蔵メモリを使う

命令メモリとデータ・メモリは，FPGAの内部メモリ・ブロックを使います．今回は，それぞれ8Kバイトのメモリ空間として作成します．命令用メモリ・モジュールのVerilog HDL記述をリスト2に示します．データ用メモリ・モジュールに関してもまったく同じ構成です．ただし，最後にincludeしているファイルの名前が異なるため，別のファイルにしています．このinst_ram_data.vとdata_ram_data.vには，メモリの初期化内容が記述されています．詳しくは後述します．

リスト1　LEDとUARTを使ったサンプル・システムのVerilog HDL記述(system_top.v)

```
module system_top (
  clk_i,
  rst_i_n,
  led_out,
  tx_out,
  rx_in);
～中略～
  assign interrupt_n[31:1] = 31'h7FFFFFFF;
  assign interrupt_n[0] = !UART_RxInt;
  lm32_top cpu_core (
   // ----- Inputs -------
   .clk_i(clk_i),
   .rst_i(rst_i),
   // From external devices
'ifdef CFG_INTERRUPTS_ENABLED
    .interrupt_n(interrupt_n),
'endif
～中略～
 );
 //============ Instruction ROM interface ==============
 asram_top                              ┐
   #(.SRAM_DATA_WIDTH(32),              │
     .SRAM_ADDR_WIDTH(32),              │
     .READ_LATENCY(1),                  │
     .WRITE_LATENCY(1),                 │
 ～中略～                                │
     .FLASH_RSTN(0))                    │
   INST_RAM (                           ├←(1)
     // Clock and reset                 │
     .clk_i(clk_i),                     │
     .rst_i(rst_i),                     │
     // Wishbone side interface         │
～中略～                                  │
   );                                   ┘

 spartan3_inst_ramb32 inst_rom32_core (
   .clk(clk_i),
   .rst_n(!rst_i),
   .we(gnd),
   .re(!inst_sram_csn),
   .be(~inst_sram_be),
   .addr(inst_sram_addr[12:2]),
   .data_in(inst_sram_data_from_cpu),
   .data_out(inst_sram_data_to_cpu));

   //============ Data RAM interface ==============
   asram_top
     #(.SRAM_DATA_WIDTH(32),
       .SRAM_ADDR_WIDTH(32),
       .READ_LATENCY(1),
       .WRITE_LATENCY(1),
～中略～
 )
   DATA_RAM (
～中略～
     .ASRAM_STB_I(D_STB_O &
       ((data_sram_addr[31:28] == 4'h2) ||
         (data_sram_addr[31:28] == 4'hF))),  ←(2)
 );

 spartan3_data_ramb32 data_ram32_core (
   .clk(clk_i),
   .rst_n(!rst_i),
   .we(!data_sram_csn &!data_sram_wen &
      (data_sram_addr[31:28] == 4'h2)),  ←(3)
   .re(!data_sram_csn),
   .be(~data_sram_be),
   .addr(data_sram_addr[12:2]),
   .data_in(data_sram_data_from_cpu),
   .data_out(data_sram_out));
  //======= LED port
  always @ (posedge clk_i)
    begin
      if(rst_i)
        begin
          led_out_reg = 32'hFFFFFFFF;
        end
       else if ((data_sram_addr[31:28] == 4'hF) &
                   (!data_sram_wen & !data_sram_csn))
       begin
         if(!data_sram_be[0])  ←(4)
           led_out_reg[7:0] = data_sram_data_from_cpu[7:0];
～中略～
     end
   end
 // data selecter in the sram side.
 assign data_sram_data_to_cpu = (data_sram_addr[31:28] == 4'hF) ?
           led_out_reg : data_sram_out;  ←(3)

 uart_core #(
   .CLK_IN_MHZ(50),        ┐
   .BAUD_RATE(115200),     ┘←(5)
   .ADDRWIDTH(5),
   .DATAWIDTH(8)
 )
 uart_module (
～中略～
   .UART_STB_I(D_STB_O &
                (data_sram_addr[31:28] == 4'h4)),
～中略～
   .INTR(UART_RxInt),
   .SIN(rx_in),
   .SOUT(tx_out)
 );

 assign D_ACK_I = (data_sram_addr[31:28] != 4'h4) ?
                    D_ACK_I_data_ram : D_ACK_I_UART;
 assign D_DAT_I = (data_sram_addr[31:28] != 4'h4) ?
                    D_DAT_I_data_ram : D_DAT_I_UART;
 assign D_RTY_I = (data_sram_addr[31:28] != 4'h4) ?
                    D_RTY_I_data_ram : D_RTY_I_UART;
 assign D_ERR_I = (data_sram_addr[31:28] != 4'h4) ?
                    D_ERR_I_data_ram : D_ERR_I_UART;
```

これらのメモリ・モジュールでは，8ビット幅のメモリ・ブロックを四つ並べることで1ワード分（32ビット幅）としています．バイト・イネーブル入力信号（be）を使ってそれぞれのバイトを選択します．

● LED出力はレジスタとして実装

LED出力は単なるレジスタとして実装しています．**リスト1**の（4）に示すように，バイト・イネーブル信号で32ビットのレジスタのうち，どのバイトに書き込むかを選択します．

ここでは32ビットで記述していますが，実際に出力するのは下位2ビットだけです．冗長なレジスタは，論理合成時に自動的に削除されます．

● CPUコアと共に提供されるUARTを活用

UARTモジュールは，LatticeMico32開発システムに含まれています．

リスト1の（5）に示すように，パラメータを指定することで通信速度が自動計算されます．CLK_IN_MHZには，このモジュールへの入力クロック周波数をMHz単位で与えます．BAUD_RATEには希望の通信速度をbps単位で指定します．今回は，50MHzのクロック入力で115,200bpsの通信速度として作成しました．

このUARTモジュールは，設定レジスタを内部に含んでいるので，プロセッサ側からソフトウェアによる初期設定も必要です．詳細は後述します．

リスト2　命令用メモリ・モジュールのVerilog HDL記述（spartan3_inst_ramb32.v）

```
module spartan3_inst_ramb32(
  clk,
  rst_n,
  we,
  re,
  be,
  addr,
  data_in,
  data_out);
～中略～
  RAMB16_S9 #(
    // Value of output RAM registers at startup
    .INIT(9'h000),
     // Ouput value upon SSR assertion
    .SRVAL(9'h000),
     // WRITE_FIRST, READ_FIRST or NO_CHANGE
     .WRITE_MODE("WRITE_FIRST")
   ) RAMB16_S9_0 (
      .DO(data_out[7:0]),   // 8-bit Data Output
      .DOP(),   // 1-bit parity Output
      .ADDR(addr),   // 11-bit Address Input
      .CLK(clk),   // Clock
      .DI(data_in[7:0]),   // 8-bit Data Input
      .DIP(gnd),   // 1-bit parity Input
      .EN(vcc),   // RAM Enable Input
      .SSR(gnd),   // Synchronous Set/Reset Input
      .WE(we & be[0])   // Write Enable Input
   );
～中略～
`include "inst_ram_data.v"
endmodule // spartan3_ramb32
```

3．ソフトウェアの開発

設計したシステムで動作するソフトウェアとして，二つのLEDを交互に点滅させて，さらに，割り込み制御で，UARTに入力された文字をループ・バックするプログラムを作ります．

ソフトウェアは，スタートアップ・ルーチンと例外処理，メイン・ルーチンと例外ハンドラ，リンカ・スクリプトの三つの部分からなります．

● スタートアップ・ルーチンと例外処理

まず，リセット入力が解除された後に実行されるスタートアップ・ルーチンを定義します．スタートアップ・ルーチンでは，例外ベクタも関連してくるので，**リスト3**のようにすべてを一つのファイルにまとめておきます．

LatticeMico32の汎用レジスタを**表2**に示します．

リセットは例外として扱われます．ハードウェアの構成のときに指定したCFG_EBA_RESETの番地から実行を開始します．このアドレスから256バイトは例外ベクタとして使われます．例外ベクタには命令列が置かれます．例外ベクタの配置を**表3**に示します．

リスト3の最初には，例外ベクタが定義されています．

リセット例外では，__startupに分岐し，スタック・ポインタを設定します．hiとloはアセンブラの疑似命令です．引き数の，それぞれ上位16ビットと下位16ビットを選択します．リセットがかかると，例外ベクタのリセット例外の部分が実行されます．スタック・ポインタが設定されて，main関数へと分岐していきます．main関数はC言語で記述されています．

リセット以外の例外は，主となる処理とは関係なく起こるイベントです．これらの例外ハンドラは処理の後，例外発生番地にリターンしなければなりません．LatticeMico32プロセッサでは，例外ハンドラからのリターン命令が，ブレークポイント例外のためのbretと，そのほかの例外のためのeretの二つあるため，**リスト3**では，前者のための例外ハンドラ__break_interruptと，後者のための

例外ハンドラ__interruptとに区別しています．

ここで紹介するプログラム例ではブレークポイント例外は用いていません．両方ともレジスタの退避を行い（save_all），C言語で書かれたinterrupt_handler関数へと分岐しています．例外の後には，退避したレジスタを復帰し（それぞれrestore_all_and_bret，restore_all_and_eret），例外ハンドラからリターンしています．

UARTモジュールから入力されている割り込み入力が"L"になると，外部割り込み例外が発生し，EBA_RESET＋0xC0番地の命令が実行されます．そして，

表2 LatticeMico32の汎用レジスタ

レジスタ名	役　割
r0	ゼロ・レジスタ
r1～r2	関数からのリターン値
r3～r8	関数への引数
r9～r25	汎用
r26(gp)	グローバル・ポインタ
r27(fp)	フレーム・ポインタ
r28(sp)	スタック・ポインタ
r29(ra)	リターン・アドレス
r30(ea)	例外アドレス
r31(ba)	ブレークポイント・アドレス

表3 LatticeMico32の例外ベクタ

アドレス	意　味
EBA_RESET	リセット例外
EBA_RESET+0x20	ブレークポイント例外
EBA_RESET+0x40	命令バス・エラー例外
EBA_RESET+0x60	ウォッチ・ポイント例外
EBA_RESET+0x80	データ・バス・エラー例外
EBA_RESET+0xA0	ゼロ除算例外
EBA_RESET+0xC0	外部割り込み例外
EBA_RESET+0xE0	システム・コール例外

リスト3　スタートアップ・ルーチン（startup.s）

```
        .text
        .extern             main
        .extern             _sp_base
#=========================================
#       Mico32 Exception Vector
#=========================================
        # Reset
        bi          __startup
        .org 0x20
        # Breakpoint
        bi          __break_interrupt
        .org 0x40
        # InstructionBusError
        bi          __interrupt
        .org 0x60
        # Watchpoint
        bi          __interrupt
        .org 0x80
        # DataBusError
        bi          __interrupt
        .org 0xa0
        # DevideByZero
        bi          __interrupt
        .org 0xc0
        # Interrupt
        bi          __interrupt
        .org 0xe0
        # SystemCall
        bi          __interrupt
        .org 0x100
__startup:
        mvhi        sp,hi(_sp_base)
        ori         sp,sp,lo(_sp_base)
        bi          main

save_all:
        addi sp, sp, -56
        /* Save all caller save registers
           onto the stack */
        sw (sp+4), r1
        sw (sp+8), r2
        sw (sp+12), r3
        sw (sp+16), r4
        sw (sp+20), r5
        sw (sp+24), r6
        sw (sp+28), r7
        sw (sp+32), r8
        sw (sp+36), r9
        sw (sp+40), r10
        sw (sp+48), ea
        sw (sp+52), ba
        /* ra needs to be moved from
           initial stack location */
        lw r1, (sp+56)
        sw (sp+44), r1
        ret

restore_all_and_bret:
        lw r1, (sp+4)
        lw r2, (sp+8)
        lw r3, (sp+12)
        lw r4, (sp+16)
        lw r5, (sp+20)
        lw r6, (sp+24)
        lw r7, (sp+28)
        lw r8, (sp+32)
        lw r9, (sp+36)
        lw r10, (sp+40)
        lw ra, (sp+44)
        lw ea, (sp+48)
        lw ba, (sp+52)
        addi sp, sp, 56
        bret

restore_all_and_eret:
        lw r1, (sp+4)
        lw r2, (sp+8)
        lw r3, (sp+12)
        lw r4, (sp+16)
        lw r5, (sp+20)
        lw r6, (sp+24)
        lw r7, (sp+28)
        lw r8, (sp+32)
        lw r9, (sp+36)
        lw r10, (sp+40)
        lw ra, (sp+44)
        lw ea, (sp+48)
        lw ba, (sp+52)
        addi sp, sp, 56
        eret

__interrupt:
        sw (sp+0), ra
        calli save_all
        calli       interrupt_handler
        bi restore_all_and_eret

__break_interrupt:
        sw (sp+0), ra
        calli save_all
        calli       interrupt_handler
        bi restore_all_and_bret
```

__interruptへと分岐し，interrupt_handler関数が実行される，というしくみです．

● C言語関数と例外ハンドラ

リスト4にmain関数と例外ハンドラ関数を示します．このファイルでは，UARTの処理とLEDレジスタ制御，例外の処理が大きな仕事です．

UARTの内部には制御レジスタがあります．図4に本稿で利用しているレジスタを示します．それぞれのアドレスは，UARTモジュールへのベース・アドレスからのオフセットでアクセスします．リスト3から呼ばれるinterrupu_handler関数は，受信データを読み出し，そのデータを送信して，ループ・バックします．

今回は，UARTでの受信割り込みを検出したときに，その受信したデータをループ・バックしてUARTに出力します．interrupt_handler関数はUARTからの受信割り込みに反応して実行します．従ってmain関数におけるLEDの点滅は停止しません．UARTからのデータのループ・バックが可能になります．例外ハンドラでは，処理された例外を無効化して，リターンしなければいけません．外部割り込み例外の無効化には，まず，周辺機能の割り込み信号をディアサートさせて（この場合，UARTから受信データを読み出す），ペンディング割り込みレジスタ（IP）のinterrupt_n入力に一致するビットに‘1’を書き込みます．

main関数では，まず，リスト4の(1)でUARTを設定しています．リスト4の(2)では，割り込みを有効にしています．まず，UARTの受信割り込みを有効にし，プロセッサの割り込みマスク・レジスタ（IM）のマスクを外し（‘1’をセットするとアンマスク），外部割り込みを割り込みイネーブル・レジスタ（IE）で有効にしています．これらのIM，IEのビットはinterrupt_n入力のビットに一致しているため，共に‘1’をセットしています．このIM，IEをセットするwriteIM関数とwriteIE関数は，インライン・アセンブラを使ってr1の内容をそのレジスタに書いています．r1は関数への引き数が渡ってきますから，valをIM，IEに書き込んでいることになります．

リスト4の(3)では，messageをUARTから出力しています．送信バイト・レジスタ（UART_THR）に1文字ずつ書き込み，送信完了を待って次の文字を書き込むループをmessageの最後まで実行しています．

リスト4の(4)は，LEDの点滅を制御しています．約0.5秒ごとに，二つのLEDが点灯・消灯を繰り返すための制御です．この部分は無限ループになっているので，点滅は止まりません．従って，UARTの受信割り込みと併用で，LEDの点滅と，UARTのループ・バックが同時に処理されるソフトウェアを作成したことになります．

リスト4　main関数と例外ハンドラ（uart_int.c）

```
#define UART_RBR *((volatile unsigned int *)0x40000000)
#define UART_THR *((volatile unsigned int *)0x40000000)
#define UART_IER *((volatile unsigned int *)0x40000004)
#define UART_LCR *((volatile unsigned int *)0x4000000C)
#define UART_LSR *((volatile unsigned int *)0x40000014)

#define LED *((volatile unsigned int *)0xF0000000)

unsigned int writeIE(unsigned int val){
  asm volatile ("wcsr IE, r1");
}

unsigned int writeIM(unsigned int val){
  asm volatile ("wcsr IM, r1");
}

unsigned int writeIP(unsigned int val){
  asm volatile ("wcsr IP, r1");
}

void interrupt_handler(){
  char temp;

  // UART RX data arrival
  if(UART_LSR & 0x1){
    temp = UART_RBR;
    writeIP(0x1);
    while(!(UART_LSR & (1<<5))) {}
    UART_THR = temp;
  }
}
```

```
#define LED_BLINK_CYCLE 252316

int main(){
  char message[] =
    "This is a message from Mico32 processor.¥n";
  int i;

  UART_LCR = (0x3 << 0) | // 8bit data      ← (1)
    (0x0 << 2) | // 1 stop bit
    (0x0 << 3); // no parity

  UART_IER = 0x1;                           ← (2)
  writeIM(0x1);
  writeIE(0x1);

  i = 0;
  while(message[i] != '¥0'){                ← (3)
    while(!(UART_LSR & (1<<5))) {}
    UART_THR = message[i];
    i++;
  }

  while(1){                                 ← (4)
    LED = ~(0x1);
    for(i=0;i<LED_BLINK_CYCLE;i++){}
    LED = ~(0x2);
    for(i=0;i<LED_BLINK_CYCLE;i++){}
  }
  return 0;
}
```

● リンカ・スクリプト（メモリ配置の定義）

ソフトウェア・コードとして記述したtextセクションやdataセクション，bssセクションなどをどのように配置するかを決定するリンカ・スクリプトを作成します．**リスト5**にリンカ・スクリプトを示します．

スタートアップ・ルーチンは，例外ベクタを含んでいるので，**リスト5**の(1)のように命令メモリの先頭に配置します．残りの命令コード（textセクション）は，**リスト5**の(2)のように，それに続く命令メモリ領域に配置します．

LatticeMico32では，命令バスとデータ・バスが完全に独立しているので，たとえ読み出し専用のデータであっても，命令メモリに配置することはできません．そこで，**リスト5**の(3)のように，読み出し専用データ（rodataセクション）と読み書きされるデータ（dataセクション）をデータ・メモリに配置します．

データ・メモリの最後には，**リスト5**の(4)のように，スタックを配置します．_sp_baseはスタートアップ・ルーチンでスタック・ポインタを初期化する際に参照されます．

● コンパイルとメモリ初期化ファイルを作る

リスト3～**リスト5**を用いて実行可能なファイルを作るには，**図5**のコマンド・ラインを実行します（p.35のコラム「GNUツール・チェーンのコンパイル」を参照）．

これで，SフォーマットとHEXフォーマットの実行形式ファイルができ上がります．これらのメモリ・イメージを前述の命令メモリとデータ・メモリの初期値に用い，コンパイルすればよいわけです．

● 実装結果とデバッグ

筆者の偏見ではありますが，これまでのフリーのCPUコ

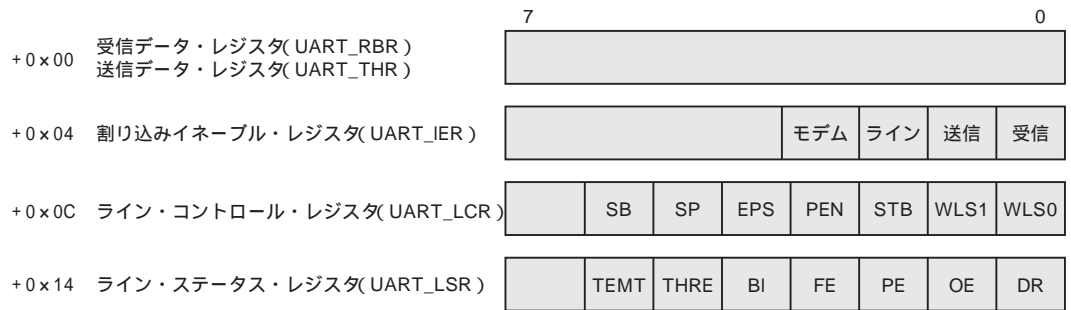



図4 UARTの制御レジスタ

コラム　GNUツール・チェーンのコンパイル

LatticeMico32のGNUツール・チェーンは，ソース・コードで提供されています．従って，利用するためにはあらかじめコンパイルして実行可能形式にする必要があります．今回は本誌付属CD-ROMにコンパイル済みのツールを収録していますが，参考までにコンパイル手順を説明します．

Mico32_gcc_src_611.tar.bz2を展開し，srcディレクトリに移動した後，**図D-1**のコマンドでコンパイルを行います．GCCのコンパイルの際にcontrib.texiでエラーが発生します．これはこのファイル中の文法が間違っているというバグです．本誌付属CD-ROMに修正済みのcontrib.texiを収録してあります．

```
#BINUTILS
cd binutils
./configure --target=lm32-elf --¥
     prefix=/usr/local/lm32-tools
make
make install
cd ..
#GCC
export PATH=/usr/local/lm32-tools/bin:$PATH
cp contrib.texi gcc/gcc/doc/contrib.texi
cd gcc
./configure --target=lm32-elf ¥
    --prefix=/usr/local/lm32-tools ¥
    --enable-languages=c
make
make install
cd ..
#newlib
cd newlib
export TARGET_CFLAGS=-DREENTRANT_SYSCALLS_PROVIDED
./configure --target=lm32-elf ¥
    --prefix=/usr/local/lm32-tools/newlib
make
make install
cd ..
#GDB
cd gdb
./configure --target=lm32-elf ¥
    --prefix=/usr/local/lm32-tools
make
make install
```

図D-1　コンパイル手順

アは「でかい」，「おそい」のそろい踏み，という印象を持っていました．しかし，LatticeMico32は，非常にコンパクトに，十分な性能が得られることが分かりました（コラム「LatticeMico32をSpartan-3Eで動作させる」とコラム「LatticeMico32をCyclone IIで動作させる」を参照）．

LatticeMico32はオープン・ソースということもあり，デバッグについても柔軟性があります．特に，ModelSimなどの論理シミュレータで，プロセッサ内部の細かな部分まで解析できます．システム開発効率という観点からも優れていると感じました．米国Mentor Graphics社のModel

コラム　LatticeMico32をSpartan-3Eで動作させる

LatticeMico32のプロセッサ・コアやその周辺機能は，Lattice Semiconductor社（以下，Lattice社）のFPGAをターゲットとし，同社の開発ツールでコンパイルすることを基本としています．デバイス・アーキテクチャに非依存でASICなどでも利用できるとされていますが，使用するツールによってはエラーになる場合が考えられます．

今回の評価に当たり，筆者の手元にLattice社のFPGAを搭載したボードが無かったため，本誌2007年7月号に付属していたSpartan-3Eボードを利用しました．FPGA開発ツールはXilinx社の「ISE」です．

ISEでLatticeMico32のVerilog HDLソース・コードをコンパイルする際には，以下の点に注意が必要です．

（1）Lattice社のマクロを回避する

lm32_addsubモジュールでは，Lattice社のマクロpmi_addsubを利用しようとします．そこでsystem_conf.vに以下の記述が必要です．

```
'define LATTICE_FAMILY "SC"
```

（2）generate文の互換性

ISEのコンパイラでは，`generate`文の中で`if`文を使う場合，`if`文に関連する`begin`節にはラベルが必要です．そこで，例えば**リストE-1**のように，`begin`の後にラベルを記述します．

（3）localparam宣言の互換性

ISEのコンパイラは，`localparam`で宣言されたパラメータ値への代入の際に，関数が渡されるとコンパイル時には自動計算できない仕様になっています．例えば，

```
localparam my_param = cond0 == value ?
value : func(arg);
```

という表現は利用できません．そこで定数を入れるような修正が必要です．load_store_unit.vとinstruction_unit.vの`addr_offset_width`に関する宣言部分で，

```
localparam addr_offset_width = 1;
```

とします．

（4）メモリ初期化のための記述

Spartan-3Eのメモリ・ブロックの初期化は，`defparam`で記述する必要があります．例えば，RAMB16_S9では，以下のような記述が必要になります．

```
defparam RAMB16_S9_0.INIT_00 = 256'h
0000000000000059
00000000000000610000000000000006d
0000000000000040;
```

ここで，`RAMB16_S9_0`はspartan3_data_ramb32.vに宣言されているRAMB16_S9のインスタンスです．

また，命令メモリとデータ・メモリのそれぞれの初期化記述が必要になります．命令とデータのバスが独立し，メモリもRAMが物理的に独立しているためです．

これらの問題を解決するために，コマンドmemcnvを作成しました．このコマンドはCygwin上で動作します．memcnvコマンドは，引き数にSフォーマットのファイルをとり，出力として，MIF，HEXフォーマット，Verilog HDLの`defparam`の記述の三つのフォーマットのファイルを出力するものです．ソース・コードと実行形式は付属CD-ROMに収録しているので参照ください．0x0から0x1FFFFFFFまでのメモリ内容が，inst_ram_dataという名前にフォーマットごとの拡張子が付いたファイルとして出力されます．0x20000000番地以降の初期値についてはdata_ram_dataという名前に拡張子の付いたファイルが出力されます．システム作成のところで触れましたが，spartan3_data_ramb32.vとspartan3_inst_ramb32.vでは，これらのVerilog HDLファイルを`include`することで，メモリの内容を初期化しています．

＊　＊　＊

これらの変更を行えば，Spartan-3Eボードで利用できるようになります．デフォルトの条件で，54%のスライス利用率で，71.4MHzで動作可能とのレポートが得られました．Spartan-3Eボードによる動作も確認できました．このコンパイル済みプロジェクトは付属CD-ROMに収録しているので，参考にしてください．ピン配置を**表E-1**に示します．

リストE-1　ラベルの記述

```
generate
      if ... begin : my_if
      ...
      end
      else begin : my_else
      ...
      end
endgenerate
```

表E-1　ピン配置

信号名	Spartan-3Eのピン番号	信号名	Spartan-3Eのピン番号
clk_i	88	tx_out	65
rst_i_n	89	led_out<0>	70
rx_in	66	led_out<1>	78

Simを使ったシミュレーションを行うためのバッチ・ファイルも本誌付属CD-ROMに収録しています．実機による検証の前に，シミュレーションの段階でソフトウェアのバグを発見できることは，今回の記事執筆中にもとても役に立った特徴でした．

参考・引用*文献

(1) LatticeMico32のページ，
http://jp.lscc.com/products/intellectualproperty/ipcores/mico32/index.html
(2) WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores,
http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf

やまぎわ・しんいち
ポルトガルINESC-ID (Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento) / Technical University of Lisbon

<筆者プロフィール>
山際伸一．ポルトガルの研究所INESC-IDのシニア研究員．博士(工学)．並列分散処理，特にクラスタ計算機の超高速ネットワーク・ハードウェアを専門とする．最近はGPUを使った高性能計算システムに関する研究を中心に多数の研究成果を発表している．CQ出版社「FPGAボードで学ぶ論理回路設計」をはじめとする書籍がある．

リスト5　リンカ・スクリプト(memory.def)

```
SECTIONS
{
        .start 0x00000000 : {  ←(1)
        _sstarttext = .;
                startup.o(.text)
        _estarttext = .;
        }
        .text : {  ←(2)
        _stext = .;
                *(.text)
        _etext = .;
        }
        .rdata 0x20000000: {  ←(3)
        _srdata = .;
                *(.rodata)
              *(.rodata.str1.4)
        _erdata = .;
        }
        .data : {
        _sdata = .;
                *(.data)
                *(.zdata)
        _edata = .;
        }
        _end = .;
        . = 0x20000000 + (1024*8) - 4;  ←(4)
        _sp_base = .;
}
```

```
$ export PATH＝$PATH:/usr/local/lm32-tools/bin
$ lm32-elf-gcc -c uart_int.c
$ lm32-elf-as startup.s -o startup.o
$ lm32-elf-ld -Map uart_int.map -T memory.def
  startup.o uart_int.o -o uart_int
$ lm32-elf-objcopy -O srec uart_int uart_int.srec
$ lm32-elf-objcopy -O ihex uart_int uart_int.hex
```

図5　実行可能ファイルを作る

コラム　LatticeMico32をCyclone IIで動作させる

米国Altera社の「Cyclone II」をターゲットに，同社のFPGA開発ツール「Quartus II」を使う際の注意点を以下にまとめます．

(1) メモリの生成

メモリは，MegaWizardで1ポート・メモリを作ってしまうのが簡単です．**図F-1**に示す1ポート・メモリを指定し，バイト・イネーブルを有効にします．Altera社のメモリ・ブロックは，MIFと呼ばれるメモリ・フォーマットか，HEXのファイルを初期化データとして使います．memcnvコマンドによりHEXフォーマットのファイルを生成して，ファイル名を初期化ファイルとして指定しておきます．

(2) Verilog HDLコードは互換性あり

Quartus IIでは，LatticeMico32のソース・コードを修正なしに利用できます．

＊　＊　＊

Cyclone II (EP2C5T144C8)をターゲットにコンパイルしたところ，約70%のロジック・エレメント使用率で80%のメモリ・ブロックを利用し，約70MHzで動作するレポートが得られました．参考として，付属CD-ROMにQuartus II 7.1で作成したプロジェクトを収録してあるので，参考にしてください．実機による動作確認は行っていませんが，シミュレーションでは正常な動作が確認できています．

図F-1　1ポート・メモリを使用する